CHAPTER

10

Developing UNIX Applications in C and C++

case ▶ Dominion Consulting's customers are becoming more conscious of computer security. The programming staff is answering an increasing number of questions about data privacy and protection. Your supervisor asks you to write a simple program that can demonstrate how file encryption and decryption works. (**File encryption** is an operation that scrambles a file's contents into a secret code. A **decryption operation** restores the file to its original state.)

L E S S O N A

o b j e c t i v e s

In this lesson you will:

- Create simple C programs
- Use the make utility to revise and maintain source files
- Identify program errors and fix them
- Create a complete C programming application

C Language Programming

In this chapter, you learn not only how to write C programs, but also how to use other software development tools, such as the make utility. The **make utility** is a UNIX program that controls changes and additions to the programs as they are being developed. Finally, you also write simple C++ programs, so you can understand how C++ programming differs from C programming.

Introducing C Programming

C is the language in which UNIX was developed and refined. The original UNIX operating system was written in assembly language. **Assembly language** is a low-level language that provides maximum access to all the computer's devices, both internal and external. However, assembly language requires more coding and a greater in-depth treatment of all internal control items. The C language was partly developed to resolve the more lengthy requirements of assembly language. It has drastically reduced these requirements to a high-level set of easy-to-understand instructions. Dennis Ritchie and Brian Kernighan, two Bell Lab employees, rewrote most of UNIX using C in the early 1970s.

Note: Ken Thompson, another Bell Lab employee, also deserves credit for his influence on the development of C. He wrote a forerunner of C, called B, in 1970 for the first UNIX system to run on the DEC PDP-7 mini-computer.

Since its inception, the C language has evolved from its original design as an operating system language to its current status as a major tool in the development of any high-performance application for general use. Since C is native to UNIX, it works best as a UNIX application development tool, where the operating system views the application as an extension of its core functionality. For example, daemons (specialized system process that run in the background) are written in C. They access the UNIX system code just as any other part of the operating system.

C programming may be described, in a nutshell, as a language that uses relatively small, isolated functions to break down large complex tasks into small and easily resolved subtasks. This function-oriented design allows programmers to create their own program functions to interact with the predefined system functions to create powerful and comprehensive solutions to the largest of applications.

Using C to write a program for Dominion's security demonstration is a good choice. Because C is a compiled language, the program source code cannot be viewed. This is an important consideration, because the source code of a security program reveals how the program works. Before you begin to write this program, however, you first need to learn the basics of C programming.

Creating a C Program

A C program consists of separate bodies of code, known as **functions**. In other languages, bodies of code have different names, such as subroutines or procedures. Each of these bodies of code is designed so it contributes to the execution of a single task. You put together a collection of these functions, and they become a program. Within the program the functions call each other as needed and work to solve the problem for which the program was originally designed.

Creating a program is never done in a single step. As a programmer, you complete many phases before the program is ready to run. The first phase is to create the source code of the program. As with shell scripts and Perl programs, you use a text editor, such as vi or Emacs, to create C programs.

The next phase is to execute the preprocessor and compiler. The **preprocessor** makes modifications to your program, such as including the contents of other files and creating constant values. After the preprocessor prepares your program, the **compiler** executes. The compiler is a program that translates the source code into **object code**, which consists of binary instructions. If you made errors, the compiler locates many of them. When this happens, you use the text editor to correct the errors and recompile the program.

Many compilers translate source code into assembly code. This requires that an **assembler** be invoked to translate the assembly code into object code. The compiler usually invokes the assembler automatically, so you do not need to enter additional commands. Some compilers translate directly from source code into object code, skipping the assembly step. Whatever type of compiler you use, the outcome of this phase is the creation of a file that contains object code.

The final phase requires the use of another tool called a **linker**. This program links all the object files that belong to the program, along with any library functions the program may use. The result is an **executable file**. The entire process is depicted in Figure 10-1.

C Key Words

The C language, like all programming languages, includes **key words**. These key words have special meanings, so you cannot use them as names for variables or functions. Table 10-1 lists C key words.

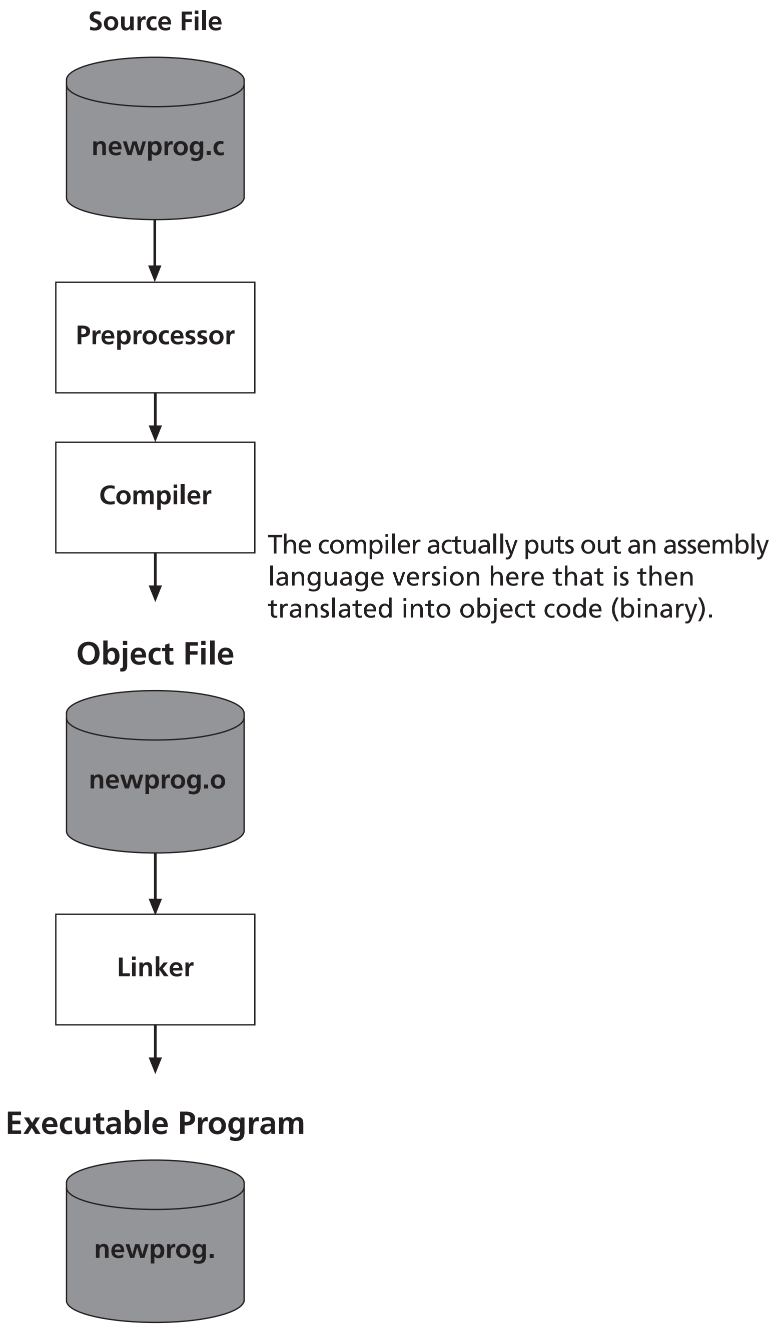

Source File
newprog.c
Preprocessor
Compiler
The compiler actually puts out an assembly language version here that is then translated into object code (binary).
Object File
newprog.o
Linker
Executable Program
newprog.

Figure 10-1: C program compilation process

The C Library

As you can see from Table 10-1, the C language is very small. It has no input or output facilities as part of the language. All I/O is performed through the C library. The **C library** consists of functions that perform file, screen, and keyboard operations, as well as many other tasks. For example, certain functions perform string operations, memory allocation and control, math operations, and much more. When you need to perform one of these operations in your program, you place a **function call** at the desired point. The linker joins the code of the library function with your program's object code to create the executable file.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 10-1: C key words

Program Format

As mentioned earlier, C programs are made up of one or more functions. Every function must have a name, and every C program must have a function called **main**. Here is a very simple C program:

```
int main()
{
}
```

This program does absolutely nothing, yet it contains all the elements necessary for a valid C program. The next two paragraphs examine the bare essentials.

Note the word "main" followed by a set of parentheses. (A following section, "Specifying Data Types," defines the first item—int.) This is the name of a function. As mentioned earlier, all C programs must have a function called main. The parentheses denote that this is a function name.

On the next line is an opening brace. In a C program this denotes the beginning of a block of code. The closing brace on the next line denotes the end of the block of code. All functions must have an opening and a closing brace. The statements that normally make up the function appear between the two braces. In the sample program there are no statements; therefore, the function does nothing. The braces are still required.

Including Comments

The /* symbol denotes the beginning of a comment, and the */ symbol denotes the end of a comment. The compiler ignores everything in between. This example shows a C program comment:

```
/* Here is a program that does nothing. */
int main()
{
}
```

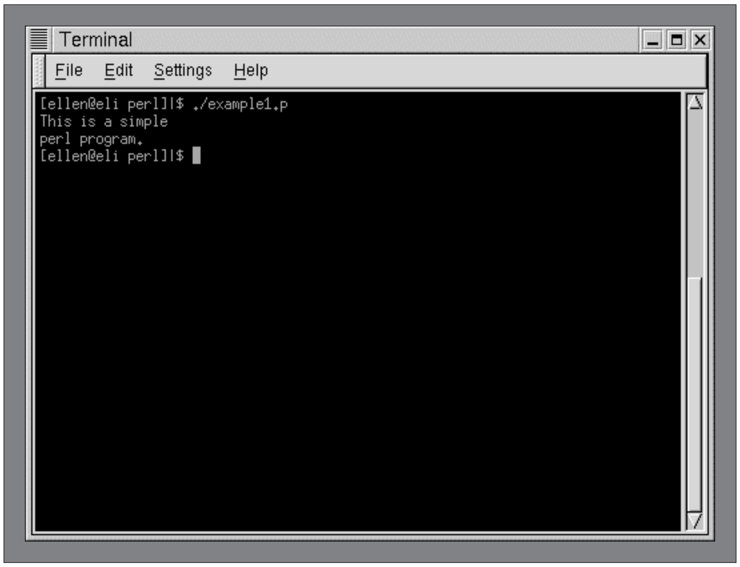
In the example, the comment "Here is a program that does nothing" appears at the top of the program. The beginning of the comment is marked with /* and the end with */. The compiler sees this program as being no different than the earlier version that had no comment.

Using the Preprocessor #include Directive

Here is a sample program that performs output:

```
/* A simple C program */
#include <stdio.h>
int main()
{
     printf("Hello from the Linux World!\n");
}
```

Figure 10-2 shows the program's output.



```
[ellen@eli perl]]$ ./example1.p
This is a simple
perl program.
[ellen@eli perl]]$
```

Figure 10-2: C program output

In the program above, you see the statement:

```
#include <stdio.h>
```

This is called a **preprocessor directive**. As mentioned earlier, the preprocessor processes your program before the compiler translates it into object code. It reads your program, looking for statements that begin with the # symbol. These statements are considered preprocessor directives and cause the preprocessor to modify your source code in some way. For example, the #include directive causes the preprocessor to include another file in your program, at the point where the #include directive appears.

The file **stdio.h** is called a **header file** and is part of your C development system. This file contains information the compiler needs to process standard input or output statements. Any program that performs standard input or output must include the stdio header file. Because the sample program uses the printf statement (which performs standard output), it must include stdio.h.

The C development system includes a number of header files. All library functions require that you include a particular header file.

Specifying Data Types

Variables and constants represent data used in a C program. You must declare variables and state the type of data that the variable will hold. A variable's data type determines the upper and lower limits of its range of values. Data types with wider ranges of values occupy more memory than those with narrower ranges. The exact limits of the ranges vary among compilers and hardware platforms.

Table 10-2 shows a list of the basic data types that may be used in a C program.

Data Type	Description
char	Occupies a single byte. Designed to hold one character from the character set used by the running machine.
int	Holds integer values. The size of an int variable should be the natural size of an integer on the running machine, but it is not always.
float	A single-precision, floating-point value.
double	A double-precision, floating-point value.

Table 10-2: C data types

As mentioned earlier, the exact upper and lower limits of each of the range of values for data types depends on the compiler and hardware platform being used. You can use three modifiers with int data types: short, long, and unsigned. The short and long modifiers make an integer variable smaller or larger than its natural size. Typically, a long int occupies twice as many bits as an int. On some machines a short int occupies half the number of bits as an int, but in many cases there is no difference between a short int and an int.

Table 10-3 shows typical limits and memory requirements of C data types.

Data Type	Bytes	Minimum Value	Maximum Value
char	1	-128	127
unsigned char	1	0	255
short int	2	-32,768	32,767
unsigned short	2	0	65,535
int	4	-2,147,483,648	2,147,483,647
long int	4	-2,147,483,648	2,147,483,647
unsigned long	4	0	4,294,967,295
float	4	-3.4028E+38	3.4028E+38
double	8	-1.79769E308	1.79769E+308

Table 10-3: Typical C data type limits and memory requirements

Character Constants

Characters are represented internally in a single byte of the computer's memory. When a character is stored in the byte, it is set to the character's code in the host character set. For example, if the machine uses ASCII codes, the letter A is stored in memory as the number 65. This is because the ASCII code for A is 65.

When you represent character data in a program as a character constant, you enclose the character in single quote marks. Here are some examples:

‘A’
‘C’
‘a’
‘z’

Using Strings

A string is a group of characters, like a name. Strings are stored in memory in consecutive memory locations. When you use string constants in your C program, they must be enclosed in double quote marks. Here are some examples:

“Linux is a great operating system.”
“Good Morning!”
“Enter your name and age.”

Unlike higher level languages, C does not provide a specific data type for character strings. C requires that you view strings the same way the computer does, as an array of characters. Here is how you might declare a character array to store a string:

```
char name[20];
```

This is just like declaring a char variable, except for the [20] appended to the variable name. It indicates that name should be an array of 20 characters. It is large enough to hold a string of up to 19 characters. This is because in C all strings are terminated with a null character. A **null character** is a single byte where all bits are set to zero.

Including Identifiers

Identifiers are names given to variables and functions. When naming variables and functions, resist the temptation to use short names that do not convey the meaning of the item. Using meaningful identifiers greatly enhances the style of your program. There are only a few rules to remember:

- The first character must be a letter or an underscore (the _ character).
- After the first character you may use letters, underscores, or digits.
- Variable names may be limited to 31 characters, and some compilers require the first 8 characters of variable names to be unique.
- Uppercase and lowercase characters are distinct.

These are all examples of legal identifiers:

radius
customer_name
earnings_for_2000
_my_name

Declaring Variables

You must declare all variables before you use them in a program. A declaration begins with a data type and is followed by one or more variable names. Here is an example:

```
int days;
```

This example declares a variable named days. Its data type is int, so days is large enough to hold any value that fits within the range of an int. Notice that the declaration ends with a semicolon, as do all complete C statements.

You can declare multiple variables of the same type on the same line. Here is an example:

```
int days, months, years;
```

This example declares three variables, each of type int, named days, months, and years. Notice that commas separate the names.

You can initialize variables with values at the time they are declared by placing an equal sign after the variable name and then a constant value. Here is an example:

```
int days = 5, months = 2, years = 10;
```

Understanding the Scope of Variables

The **scope** of a variable is the part of the program in which the variable is defined and therefore accessible. You can declare a variable either inside a function or any place that is not inside a function.

Variables that are declared inside a function are called **automatic variables**. These variables are local to the function in which they are declared. Here is an example:

```
/* This program declares a local variable
   in function main. The program does nothing
   else.  */
int main()
{
int days;
}
```

Here, the variable days is an automatic variable and is local to the function main.

You can also declare a variable outside of any function, as in the following example:

```
/* This program declares a global variable
   The program does nothing else.  */

int days;
int main()
{
}
```

In the program above, the variable is external, or global. The scope of a global variable is the entire program, beginning at the point where the declaration was made. The scope of an automatic, or local, variable is the body of the function in which it is declared.

The only place inside a function where local variables may be declared is at the beginning of the body of the function—after the opening brace and before any statement. You can declare global variables anywhere in a program except inside a function.

Using Math Operators

Table 10-4 lists the C arithmetic operators.

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus
++	increment
--	decrement

Table 10-4: C arithmetic operators

You can use these operators to create regular math expressions, as in the following examples:

```
x = y + 3;
num = num * 3;
days = months * 30;
```

These examples introduce the assignment operator (the equal sign). It works by taking the value of the expression on its right and assigning that value to the variable whose name is on its left. In the example days = months * 30, the value in the variable months is multiplied by 30 and the product is stored in the variable days.

Increment and Decrement Operators The last two operators shown in Table 10-4 are the **increment** (++) and **decrement** (--) operators. These are unary operators, meaning they work with one operand. The following example shows the variable count being incremented:

```
count++;
```

Likewise, this variable can be decremented by the following statement:

```
count--;
```

The first two examples of the count variable show these operators in their postfix form, which means they come after the variable. You can also use them as prefix operators:

```
++ count;
-- count;
```

The operators behave differently depending on which form is used. Assume the variable j is set to 4. In this statement,

```
x = j++;
```

the ++ operator is used in postfix form. This means the assignment operator (=) uses the value of j before it is incremented. In effect, it says "set x equal to j, then increment j." After the operation, x will be equal to 4 and j will be equal to 5.

If the prefix form of the operator is used, you get different results:

```
x = ++j;
```

This statement says "increment j, then set x equal to j." Both x and j will be equal to 5 after the statement executes.

Generating Formatted Output with printf

One of the most commonly used screen output library functions is printf. The f stands for "formatted," as the function allows you to format and print several arguments of differing data types. The printf function is used in the following manner:

Syntax printf (control string, expression, expression,...)

The first argument is called the **control string**. It specifies the way formatting should occur. Following the control string may be a variable number of arguments. Each of these will be an expression whose value is to be printed. Here is perhaps the most simple example of a printf statement:

```
printf("Hello");
```

The example only uses a control string. The word Hello will be printed as is on the screen. Here is another example.

```
printf("Your age is %d", 30);
```

The %d that appears in the control string is called a format specifier. It will not be printed as part of the message, but tells printf to substitute a decimal integer in its place. The decimal integer is the very next argument, the number 30. This printf statement prints the following message on the screen:

```
Your age is 30
```

Although this example illustrates the usage of the %d format specifier, it is not very realistic. You are more likely to use it in the following manner:

```
printf("Your age is %d", age);
```

Here, printf substitutes the value in the integer variable age for the %d. The next example prints the values of three int variables:

```
printf("The values are %d %d %d", num1, num2, num3);
```

This message will contain the values of num1, num2, and num3, in that order. You can also pass arithmetic expressions to printf:

```
printf("You have worked %d minutes", hours*60);
```

In fact, you can pass any valid C expression to printf. However, be sure you use an appropriate format specifier. Table 10-5 shows a list of valid format specifiers.

Format Specifier	Meaning
%c	Single character
%d	Signed decimal integer
%e	Floating-point number, e notation
%E	Floating-point number, E notation
%f	Floating-point number, decimal notation
%g	Causes %f or %e to be used, whichever is shorter
%G	Causes %f or %E to be used, whichever is shorter
%i	Signed decimal integer
%o	Unsigned octal integer
%p	Pointer
%s	Character string
%u	Unsigned decimal integer
%x	Unsigned hex integer using digits 0-f
%X	Unsigned hex integer using digits 0-F
%%	Print a percent sign

Table 10-5: Format specifiers

You have learned enough C programming basics to write a simple program.

To write a simple C program:

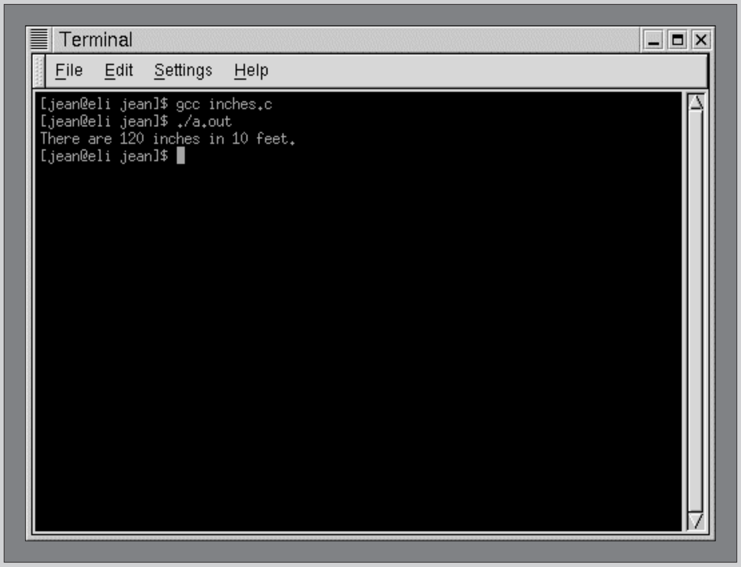
1 Use the editor of your choice to create the file **inches.c**. Enter the code:

```
/* This program converts 10 feet to inches. */

#include <stdio.h>

int main()
{
     int inches, feet;
     feet = 10;
     inches = feet * 12;
     printf("There are %d inches in %d feet.\n", inches,
feet);
}
```

2 Save the program and exit the editor.

3 The C compiler is executed by the gcc command. Type **gcc inches.c** and press **Enter**. If you typed the program correctly, you see no messages. If you see error messages, load the program into the editor and correct the mistake.

4 By default, the compiler stores the executable program in a file named a.out. Execute a.out by typing **./a.out** and pressing **Enter**. Your screen looks similar to Figure 10-3.



```
[jean@eli jean]$ gcc inches.c
[jean@eli jean]$ ./a.out
There are 120 inches in 10 feet.
[jean@eli jean]$
```

Figure 10-3: Output of a.out

5 You can specify the name of the executable file with the –o option. Type **gcc –o inches inches.c**, and press **Enter**. The command compiles the Inches.c file and stores the executable code in a file named Inches.

6 Run the inches program by typing **./inches** and pressing **Enter**.

Using the if Statement

The if statement allows your program to make decisions depending upon whether a condition is true or false. The general form of the if statement is:

Syntax if (condition) statement;

If the condition is true, statement will be performed. Here is an example:

```
if (weight > 1000) printf("You have exceeded the limit.");
```

If the variable weight contains a value greater than 1000, the printf statement executes.

Sometimes you may need to execute more than one line of code if a condition is true. C allows you to substitute a block of code for the single statement, when necessary. Here is an example:

```
     if (weight > 1000)
     {
        printf("Warning!\n");
        printf("You have exceeded the limit.\n");
        printf("Just thought you\'d like to know.");
     }
```

The program segment above causes the three printf statements to execute if weight is greater than 1000.

The if-else construct allows your program to do one thing if a condition is true and another if it is false. Here is an example:

```
if (hours > 40)
     printf("You can go home now");
else printf("Keep working.");
```

The “keep working” message prints only when the condition (hours > 40) is false. Here is an example using blocks of code:

```
if(hours>40)
{
     printf("Go home.\n");
     printf("You deserve it.");
}
else
{
     printf("Keep working.\n");
     printf("Stop playing with the computer.");
}
```

To practice the C if-else statement:

1 Create the file **radius.c** with your choice of editor. Enter the following C code:

```
/* This program calculates the area of a circle */
#include <stdio.h>
int main()
{
     float radius = 50, area;
     area = 3.14159 * radius * radius;
     if (area > 100)
          printf("The area, %f,  is too large.\n", area);
     else
          printf("The area, %f, is within limits.\n",
area);
}
```

2 Save the file and exit the editor.

3 Compile the program by typing **gcc –o radius radius.c** and pressing **Enter**. If you see error messages, edit the file and correct your mistakes.

4 Execute the program by typing **./radius** and pressing **Enter**.

Using C Loops

Loops in C are similar to those you have used in shell scripts and Perl programs. C provides three looping mechanisms: the for loop, the while loop, and the do while loop. Using the for loop is best when you know how to control the number of times that the loop is to perform. If it is unclear how many times the loop should perform, then use the while or do-while loop.

Here is an example of the for loop:

```
for (count = 0; count < 100; count++)
     printf("Hello\n");
```

This loop means the message “Hello” will print 100 times. Following the word “for” is a set of parentheses containing three arguments. The arguments are separated by semicolons.

The first argument is the initialization. The variable count is being used to track the number of times the loop has run. The initialization is a statement that is executed before the first time through. In the example above, the initialization stores the number 0 in count.

The second argument is the test condition. The for loop executes as long as the test condition is true. It is evaluated before each iteration of the loop. If the condition is true, the iteration is performed. Otherwise, the loop terminates. In the example, the loop performs as long as count is less than 100.

The third argument is the update. It is performed at the end of each iteration. In the example, the loop increments the variable count.

This program segment shows an example of the while loop:

```
x=0;
while (x++ < 100)
     printf("x is equal to %d\n", x);
```

This loop repeats while x is less than 100. The next example illustrates a do-while loop, which is very similar to the while loop.

```
x=0;
do
     printf("x is equal to %d\n", x);
while (x++ < 100);
```

The difference between the while loop and the do-while loop is that the while loop tests its condition before each iteration, and the do-while loop tests after each iteration.

To practice using a C loop:

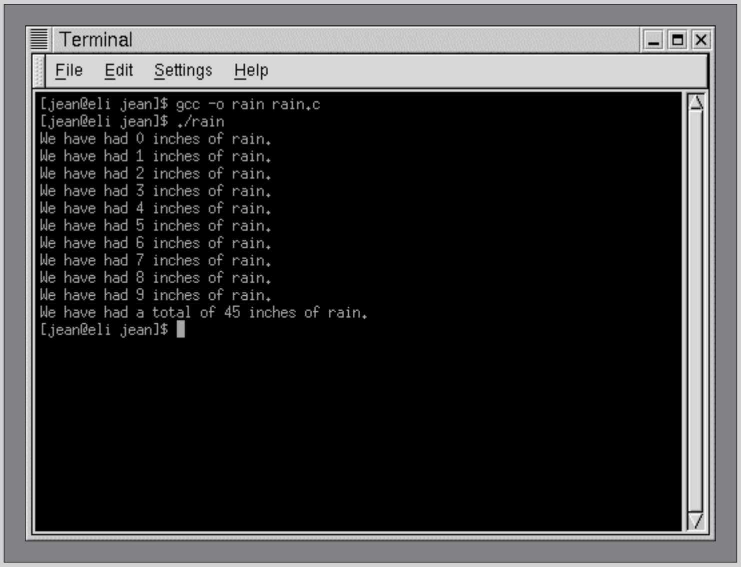
1 Use the editor of your choice to create the file **rain.c**. Enter this C code:

```
/* rain.c  */
#include <stdio.h>
int main()
{
     int rain, total_rain = 0;
     for (rain = 0; rain < 10; rain++)
     {
          printf("We have had %d inches of rain.\n", rain);
          total_rain = total_rain + rain;
     }
     printf("We have had a total/#”);
     printf(“/of %d inches of rain.\n", total_rain);
}
```

2 Save the file and exit the editor.

3 Compile the program and store the executable code in a file named rain.

4 Run the program. Your screen should look similar to Figure 10-4.



```
[jean@eli jean]$ gcc -o rain rain.c
[jean@eli jean]$ ./rain
We have had 0 inches of rain.
We have had 1 inches of rain.
We have had 2 inches of rain.
We have had 3 inches of rain.
We have had 4 inches of rain.
We have had 5 inches of rain.
We have had 6 inches of rain.
We have had 7 inches of rain.
We have had 8 inches of rain.
We have had 9 inches of rain.
We have had a total of 45 inches of rain.
[jean@eli jean]$
```

Figure 10-4: Output of rain

Defining Functions

When you define a function, you declare the function's name and create the lines of code that make up the function's block of code. You also state what data type is returned from the function (if any). Here is an example.

```
void message()
{
     printf("Greetings from the function message.");
     printf("Have a nice day.");
}
```

The word "void" indicates that this function does not return a value. The name of the function is message. A set of parentheses follows the name. There are only two statements in this function, both printfs. The function might appear in a complete program as

```
#include <stdio.h>

void message();

int main()
{
     message();
}
```

```
void message()
{
     printf("Greetings from the function message.\n");
     printf("Have a nice day.\n");
}
```

The line under the include statement that reads,

```
     void message();
```

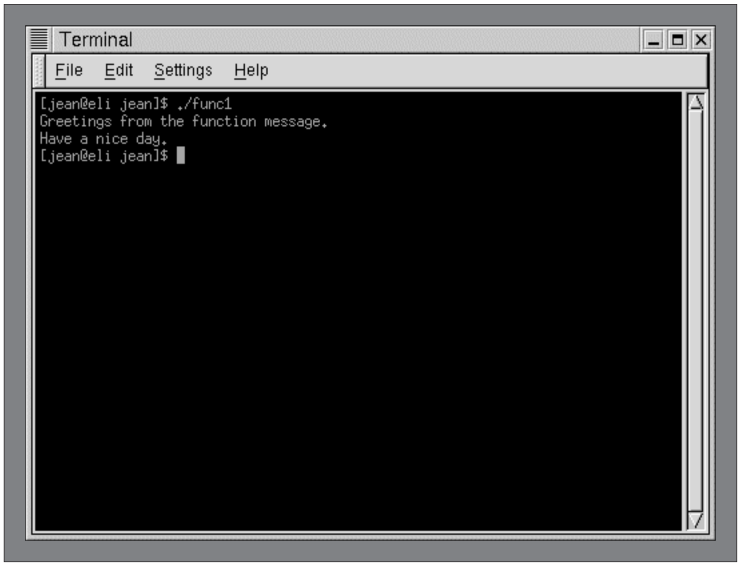
is called a function prototype. It tells the compiler about the function in advance. The word "void" means that this function returns no data. Void functions in C are like subroutines in Fortran or procedures in Pascal. They are merely modules of code that perform some task.

After the function prototype comes the function main. Main includes only one line, which reads:

```
     message();
```

This is a function call. You call functions by placing their name, followed by a set of parentheses and a semicolon, at the desired place in the program. This causes the program's control to pass to the function. When the program returns from the function, it resumes execution at the very next line after the function call.

After main is the definition of the function message. The output of the program is shown in Figure 10-5.



```
[jean@eli jean]$ ./func1
Greetings from the function message.
Have a nice day.
[jean@eli jean]$
```

Figure 10-5: Demonstration of a function

Using Function Arguments

Sometimes it is necessary to pass information to a function. A value passed to a function is called an argument. Arguments are stored in special automatic variables.

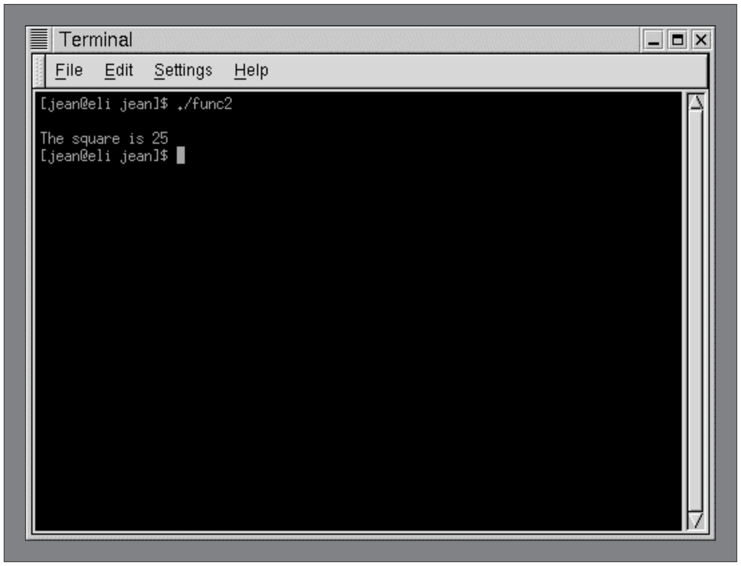
Here is an example.

```
void print_square(int val)
{
     printf("\nThe square is %d", val*val);
}
```

This function takes an int argument. When it receives the argument, the function stores the argument in the variable val. The printf statement causes the value of the expression val*val to print. Here is a complete program that uses the function:

```
#include <stdio.h>

     void print_square(int val);

main()
{
     int num = 5;
     print_square(num);
}


void print_square(int val)
{
     printf("\nThe square is %d\n", val*val);
}
```

The output of the program is shown in Figure 10-6.



```
[jean@eli jean]$ ./func2

The square is 25
[jean@eli jean]$ 
```

Figure 10-6: Demonstration of a function argument

Using Function Return Values

In addition to accepting arguments, functions may also return a value. This means you can make function calls part of arithmetic operations and assignments. For example, suppose you have a function called triple. It is designed to take an int argument and return that value multiplied by three. You could use the function call in a manner such as:

```
y = triple(x);
```

The function receives the value in x, triples it, and then returns this value. The statement above stores the return value in a variable called y. Here is what the triple function might look like:

```
     int triple(int num)
     {
          return(num * 3);
     }
```

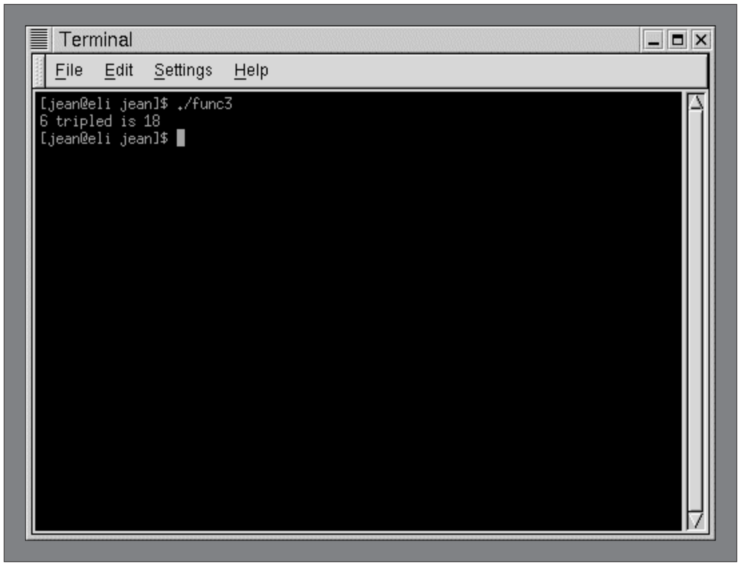
The function is defined as an int function. This means that it returns an int value. You may place a call to this function anywhere in your program where an int is expected. The function takes a single argument, which is also an int. In the function the argument is stored in the variable num. There is only one line in the function's block of code:

```
return(num*3);
```

This is the return statement. It is used to return a value back to the calling part of the program. In this example the value of num * 3 is returned. The next sample program demonstrates the function:

```
#include <stdio.h>

int triple(int num);

int main()
{
     int x = 6, y;
     y = triple(x);
     printf("%d tripled is %d\n.", x, y);
}

int triple(int num)
{
     return (num * 3);
}
```

The program output is shown in Figure 10-7.



```
[jean@eli jean]$ ./func3
6 tripled is 18
[jean@eli jean]$
```

Figure 10-7: Demonstration of a program's return value

To practice writing functions that accept arguments and return a value:

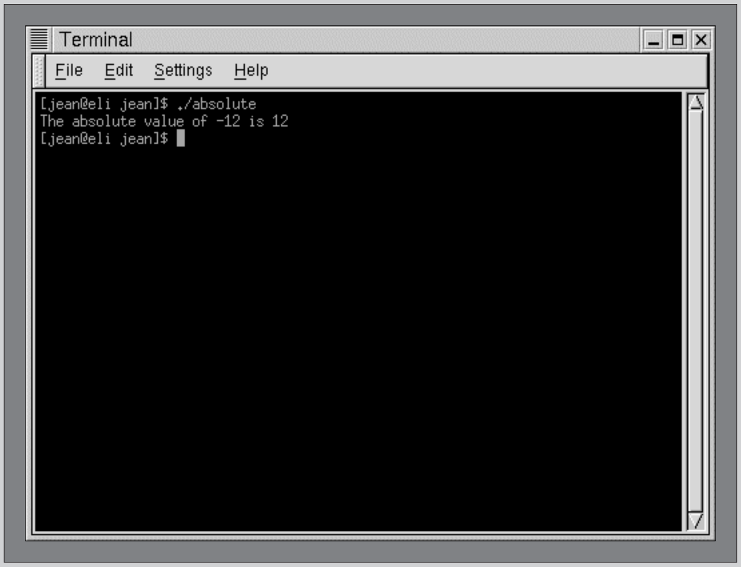
1 Use the editor of your choice to create the file **absolute.c**. Enter the following code:

```
#include <stdio.h>

int absolute(int num);
int main()
{
     int x = -12, y;
     y = absolute(x);
     printf("The absolute value of %d is %d\n", x, y);
}

int absolute(int num)
{
     if (num < 0)
          return (-num);
     else
          return (num);
}
```

2 Save the file and exit the editor.

3 Compile the program and save the executable code in a file named **absolute**.

4 Run the program. Your screen should look similar to Figure 10-8.



```
[jean@eli jean]$ ./absolute
The absolute value of -12 is 12
[jean@eli jean]$
```

Figure 10-8: Output of absolute program

Working with Files in C

Files are continuous streams of data. They are typically stored on disk. Many file operations are sequential, meaning they work from the beginning of the file to the end. When the file is opened, you are working with the beginning of the file. Every time a byte is read from or written to the file, your current position in the file is moved forward by one byte.

File Pointers C file input/output is designed to use file pointers, which point to a predefined structure that contains information about the file. The structure template is found in stdio.h. You must declare a file pointer in order to use the I/O package. Here is an example:

```
FILE *fp;
```

This declares fp as a FILE pointer. It will be used with various file access functions.

Opening and Closing Files Before you can use a file, it must be opened. The library function for opening a file is fopen. Here is an example.

```
fp = fopen("myfile.dat", "r");
```

The fopen function takes two arguments: the filename and the access mode. This example opens a file named myfile.dat. The “r” means that the file will be opened for reading. The following statement uses the “w” access mode for writing:

```
fp = fopen("myfile.dat", "w");
```

The fopen function returns a file pointer. If the file cannot be opened, it returns a NULL pointer (a pointer to address zero). Here is one way you can test to see if the file was opened:

```
if ((fp = fopen("myfile.dat", "r")) == NULL)
{
     printf("Error opening myfile.dat\n");
     exit(0);
}
```

The opposite of opening a file is closing it. When a file is closed, its buffers will be flushed, ensuring that all data was properly written to it. The fclose function is used to close files that were opened by fopen. Here is an example:

```
fclose(fp);
```

Performing File Input/Output C provides many functions for reading and writing files. For the case project, you concentrate on two: fgetc and fputc.

The two functions, fgetc and fputc, perform character input/output on files. Here is an example of fgetc:

```
ch = fgetc(fp);
```

fgetc reads a single character from the file and points to it. The character will be read from the current position. Character output is performed with fputc. Here is an example:

```
fputc(ch, fp);
```

The character stored in ch is written to the current position of the file referenced by fp.

Testing for the End of File Use the feof function to determine if the end of file has been encountered during an input operation. Here is an example:

```
          if (feof(fp))
               fclose(fp);
```

The feof function returns a non-zero value if the end of file was encountered. Otherwise, it returns 0.

Now that you have a basic understanding of file operations in C, you are ready to practice writing a program that performs file input/output:

To perform file input/output:

1 Use the editor of your choice to create the file **buildfile.c**. Enter this code in the file:

```
#include <stdio.h>
```

```
int main()
{
     FILE *out_file;
     int count = 0;
     char msg[] = "This was created by a C program.\n";

     if ((out_file = fopen("testfile", "w")) == NULL)
     {
          printf("Error opening file.\n");
          exit(1);
     }
     while (count < 33)
     {
          fputc(msg[count], out_file);
          count++;
     }
     fclose(out_file);
}
```

2 Save the program and exit the editor.
3 Compile the program and save the executable in a file named **buildfile**.
4 Run the **buildfile** program. The program creates another file, testfile.
5 To see the contents of testfile, use the **cat** command.

Using the Make Utility to Maintain Program Source Files

You may often work with a program that has many source-code files. For example, the absolute program you created in the previous exercise can be divided into two files: one that holds the function main and another that holds the function absolute. The two files are then compiled and linked together, as demonstrated in the following steps.

To compile and link two files:

1 Use the editor of your choice to create the file **abs_func.c**. Enter this code:

```
int absolute(int num)
{
     if (num < 0)
          return (-num);
     else
          return (num);
}
```

2 Save the file.

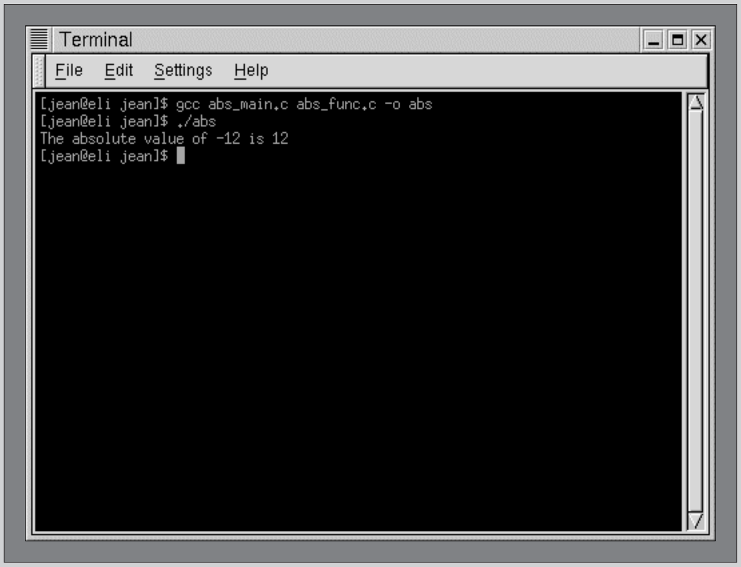
3 Create the file **abs_main.c**. Enter this code:

```
#include <stdio.h>

int absolute(int num);
int main()
{
     int x = -12, y;
     y = absolute(x);
     printf("The absolute value of %d is %d\n", x, y);
}
```

4 Save the file and exit the editor.

5 Compile and link the two programs by typing **gcc abs_main.c abs_func.c –o abs** and pressing **Enter**. The compiler separately compiles abs_main.c and abs_func.c. Their object files are linked together, and the executable code is stored in the file abs.

6 Run the abs program. Your screen should look similar to Figure 10-9.



```
[jean@eli jean]$ gcc abs_main.c abs_func.c -o abs
[jean@eli jean]$ ./abs
The absolute value of -12 is 12
[jean@eli jean]$ 
```

Figure 10-9: Output of abs program

As you develop multi-module programs and make changes, you must compile the program repeatedly. However, with multi-module source files, you only need to compile those source files in which you made changes. The linker then links the newly generated object-code files with previously compiled object-code, thereby creating a new executable file. However, keeping track of what needs to be recompiled and what does not can become an overwhelming task when the program involves dozens of source-code files. This is where the make utility helps.

The **make utility** tracks what needs to be recompiled by using the time stamp field, which is stored in all the source files. All you have to do is create a control file, called the **makefile** (which is actually a file named makefile), for the make utility to use. The control file lists all your source files and their relationships to each other. These relationships are expressed in the form of targets and dependencies. A **target file** depends on another file to determine if any action needs to be taken to rebuild the target file. (The ultimate target file is, of course, the executable file that results from linking all the object files together.) The **dependent files** are source files, such as the .c source files, or .h files that serve as headers to be included within the source files.

The makefile must exist in the current directory. It feeds the make utility all it needs to know to first examine and recompile any changed modules and then relink the objects to produce a new executable program. You can also give the makefile another name, such as make_abs. To do this, you need to enter an –f option followed by the name of the makefile. This is useful when you are developing more than one application from within the same directory.

The contents of make_abs, an example of a makefile, follow:

```
abs_main.o: abs_main.c
     gcc –c abs_main.c
abs_func.o: abs_func.c
     gcc –c abs_func.c
abs2: abs_main.o abs_func.o
     gcc abs_main.o abs_func.o –o abs2
```

Two types of lines are shown in the file: dependencies and commands. The first line is a dependency, and the second line is a command:

```
abs_main.o: abs_main.c
     gcc –c abs_main.c
```

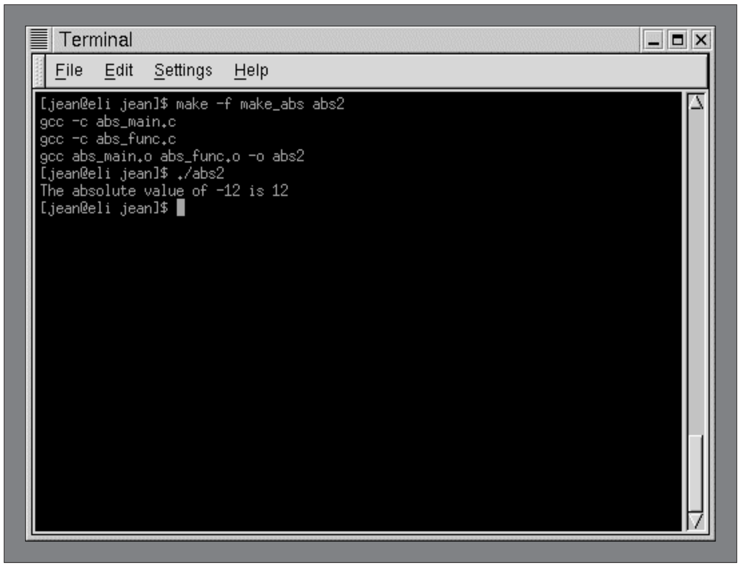
The first line establishes a dependency between abs_main.o and abs_main.c. If abs_main.c is newer than abs_main.o, the command on the second line executes (rebuilding abs_main.o).

The third and fourth lines, as well as the fifth and sixth lines, establish similar dependencies and commands.

The command line entry to build the abs2 program using the makefile is:

```
make –f make_abs abs2
```

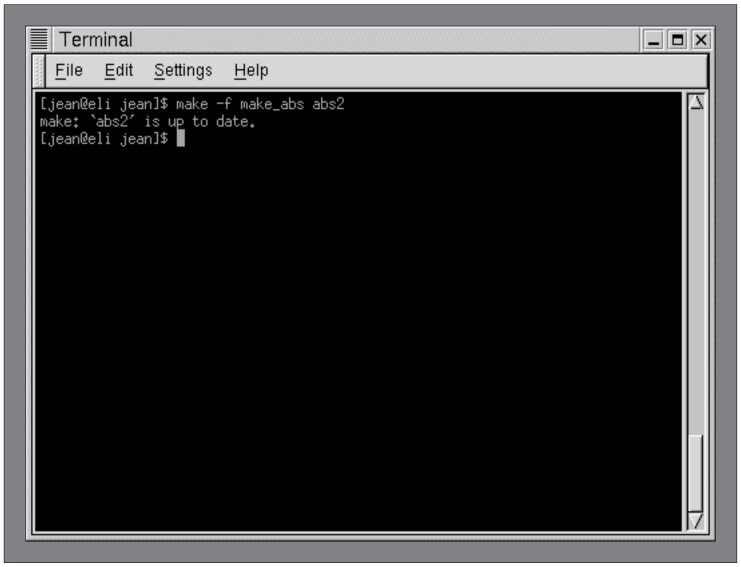
The –f option instructs make to read the file make_abs instead of makefile. After executing the command, you can run the abs2 program. Figure 10-10 shows the output of the make command and the abs2 program.



```
[jean@eli jean]$ make -f make_abs abs2
gcc -c abs_main.c
gcc -c abs_func.c
gcc abs_main.o abs_func.o -o abs2
[jean@eli jean]$ ./abs2
The absolute value of -12 is 12
[jean@eli jean]$ 
```

Figure 10-10: Making and running abs2 program

If you forget whether you have made changes since the last time you ran the program, you can use make to check the source files' time stamps and rebuild the program if necessary. The make utility will not recompile if the program is current. For example, Figure 10-11 shows the output of the make command when all modules of the abs2 program are up to date.



```
[jean@eli jean]$ make -f make_abs abs2
make: `abs2' is up to date.
[jean@eli jean]$ 
```

Figure 10-11: Output of make when all modules are up to date

The make utility follows a set of rules, both defaults and user-defined. In general, a make rule has:

- A target, the name of the file you want to make (in the example above, the target is Abs2)
- One or more dependencies, the files upon which the target depends
- An action, a shell command that creates the target.

Now that you have learned the structure of a makefile, you can create a simple multi-module C project.

To create a simple multi-module C project:

1 Use the editor of your choice to create the file **square_func.c**. Enter this code in the file:

```
int square(int number)
{
     return (number * number);
}
```

2 Save the file.

3 Next create the file **square_main.c**. Enter this code:

```
#include <stdio.h>
int square(int number);

int main()
{
     int count, sq;
     for (count = 1; count < 11; count++)
     {
          sq = square(count);
          printf("The square of %d is %d\n", count, sq);
     }
}
```

4 Save the file.

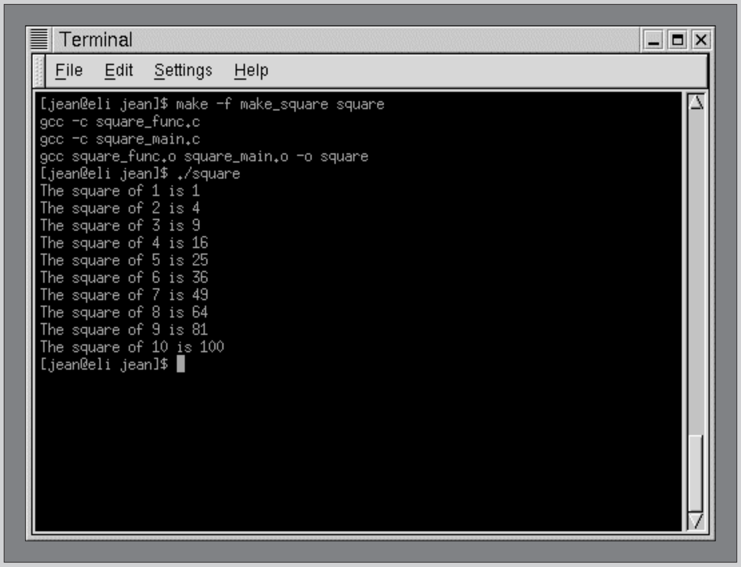
5 Next, create a makefile named **make_square**. Enter the following text:

```
square_func.o: square_func.c
     gcc —c square_func.c
square_main.o: square_main.c
     gcc —c square_main.c
square: square_func.o square_main.o
     gcc square_func.o square_main.o —o square
```

6 Save the file and exit the editor.

7 Build the program by typing **make –f make_square square** and pressing **Enter**. (If you have errors, load the incorrect module into the editor and correct your mistakes.)

8 Run the square program. Your screen should look similar to Figure 10-12.



```
[jean@eli jean]$ make -f make_square square
gcc -c square_func.c
gcc -c square_main.c
gcc square_func.o square_main.o -o square
[jean@eli jean]$ ./square
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
[jean@eli jean]$
```

Figure 10-12: Building and running the square program

Now that you understand the basics of writing a program in C, you'll learn how to debug a program.

Debugging Your Program

Typical errors for new C programmers include using incorrect syntax, such as forgetting to terminate a statement with a semicolon. Or, because almost everything you type into a C program is in lowercase, your program may have a case-sensitive error. Here is an example of what you might see on the screen if you omit a closing quote inside a printf command:

```
simple.c:10: unterminated string or character constant
simple.c:10: possible real start of unterminated constant
```

The compiler generally produces more error lines than the number of mistakes it finds in the code. The compiler reports the error lines and any surrounding lines affected by the mistake(s).

Note: Remember that every time you modify (correct or add text to) your program source file, you must recompile the program to create a new executable program.

To correct syntax errors within your programs, you must therefore perform the following steps:

1. Write down the line number of each error and a brief description.
2. Edit your source file, moving your cursor to the first line number the compiler reports.
3. Within the source file, correct the error and then move the cursor to the next line number. Most editors display the current line number to help you locate specific lines within the file.
4. After correcting all the errors, save and re-compile the file.

Now that you understand how to write and debug simple C programs, you are ready to create interactive programs that read input from the keyboard.

Creating a C Program to Accept Input

You can draw from many standard library functions to accept input; that is, enter characters using the keyboard. Some like getchar() are character-oriented, while others like scanf() are field-oriented. This section concentrates on scanf().

Unlike many other library input functions, scanf can be used to input values of a variety of data types. You use it like this:

Syntax scanf(control string, address, address,...);

The scanf() function uses a control string with format specifiers in a manner similar to printf. The arguments that follow the control string are the addresses of variables where the input is to be stored. Consider the following example.

```
scanf("%d", &age);
```

The %d format specifier works just like it does for printf. Here it indicates that scanf() should interpret the input value as a decimal integer.

The &age argument tells scanf to store the input value in the variable age. The & is the address operator. When used with a general variable, it returns the memory address where this variable is located. The scanf() function needs the address of a variable to store an input value there. The next example shows how scanf() can be used to input a string.

```
scanf("%s", city);
```

Notice that this example does not use the & operator. Anytime you use the name of an array, it resolves to the address of the first element. It would be an error to use the & operator with the name of an array.

The format specifiers for scanf() are generally the same as those used with printf(). Table 10-6 shows the format specifiers for scanf().

Format Specifier	Interpretation
%c	Single character
%d	Signed decimal integer
%e, %f, %g	Floating-point number
%E, %G	Floating-point number
%I	Signed decimal integer
%o	Signed octal integer
%p	Pointer
%s	String. Ignores leading white-space characters, then reads until it encounters another white-space character.
%u	Unsigned decimal integer
%x, %X	Signed hex integer

Table 10-6: scanf() format specifiers

Table 10-7 shows a list of modifiers you can use with scanf format specifiers.

Modifier	Meaning
h	Used to indicate a short int or short unsigned int. Example: "%hd"
l	Used to indicate a long int or long unsigned int. Example: "%ld" Also used to indicate a double. Example: "%lf"
L	Used to indicate a long double. Example: "%Lf"

Table 10-7: Modifiers for scanf() format specifiers

Although it rarely contributes to a program's user-friendliness, the scanf statement can accept multiple inputs. Here is an example:

```
scanf("%d %f %d", &x, &y, &z);
```

The statement above accepts values in the variables x, y, and z which are int, float, and int, respectively. While typing values, the user must separate the three values with white-space characters. White-space characters are spaces, tabs, and new-lines.

You'll now write a C program to accept input from a keyboard.

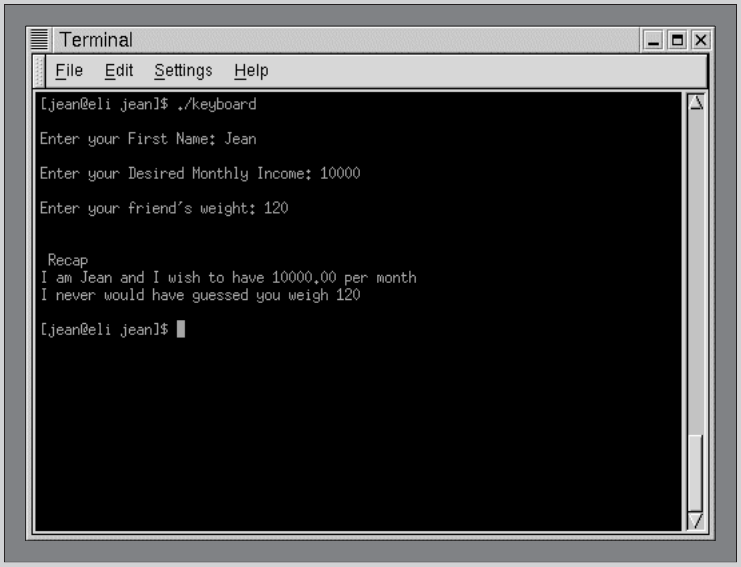
To use C to accept keyboard input:

1 Use the editor of your choice to create a file named **keyboard.c**. Enter this code and then save the file:

```
/*==========================================================
Program Name: keyboard.c
Purpose:    Enter data using the keyboard

========================================================== */

#include <stdio.h> /* the standard input/output library */
int main()
  {
   char string[50]; /* a string field */
   float my_money; /* a floating decimal field */
   int weight; /* an integer field */
   printf("\nEnter your First Name: ");
   scanf("%s", string);
   printf("\nEnter your Desired Monthly Income: ");
   scanf("%f",&my_money);
   printf("\nEnter your friend's weight: ");
   scanf("%d",&weight);
   printf("\n\n Recap\n");
   printf("I am %s and I wish to have %8.2f per month",
       string, my_money);
   printf("\nI never would have guessed you weigh %d",
      weight);
   printf("\n\n");
  }
```

2 Compile the program by typing **gcc keyboard.c –o keyboard** and then pressing **Enter**.

3 Execute the program by typing **./keyboard** and then pressing **Enter**. Your screen should now look similar to Figure 10-13. Note that your screen will appear differently depending on what you have input.



```
[jean@eli jean]$ ./keyboard

Enter your First Name: Jean

Enter your Desired Monthly Income: 10000

Enter your friend's weight: 120


 Recap
I am Jean and I wish to have 10000.00 per month
I never would have guessed you weigh 120

[jean@eli jean]$
```

Figure 10-13: Output of keyboard program

Now that you have created C programs that perform keyboard and file I/O, you are ready to write the security demonstration programs.

Encoding and Decoding Programs

If a file contains sensitive information, you may wish to encrypt it so others cannot read its contents. When a file is **encrypted**, its contents are encoded or modified in such a way that the original contents are not distinguishable. A formula is used to perform the encryption, so that an alternative **decryption** algorithm can restore the file to its original contents.

The program you have been asked to write for Dominion Consulting's programming staff will be simple in design. It will open a file, read a character from the file, add 10 to the character's ASCII value, and then write the character to a second file. This procedure repeats until all characters in the file have been read, modified, and written to the second file. The second file will be an encoded version of the first file.

The decoding program will work opposite of the way the encoding program works. It will read a character from the encrypted file, subtract 10 from its ASCII code, and write the character out to another file. This procedure repeats until all encrypted characters have been converted to their original state and stored in the second file.

To create the encoding program:

1 Use the editor of your choice to create the file **encode.c**. Enter this code in the file:

```
#include <stdio.h>

void encode(FILE *, FILE *);

int main()
{

   FILE *in_file, *out_file;
   char infile_name[81], outfile_name[81], input;

   printf("Enter the name of the file to encode: ");
   scanf("%s", infile_name);
   if ((in_file = fopen(infile_name, "r") ) == NULL)
   {
          printf("Error opening %s\n", infile_name);
          exit(0);
   }
   printf("Enter the output file name: ");
   scanf("%s", outfile_name);
   if ((out_file = fopen(outfile_name, "w") ) == NULL)
   {
          printf("Error opening %s\n", outfile_name);
          exit(0);
   }
   encode_file(in_file, out_file);
   printf("The file has been encoded.\n");
   fclose(in_file);
   fclose(out_file);
}
```

2 Save the file.

3 Create the file **encode_file.c** and enter this code:

```
#include <stdio.h>

void encode_file(FILE *in_file, FILE *out_file)
{
   char input;

   while (!feof(in_file))
   {
     input = fgetc(in_file);
          input += 10;
             fputc(input, out_file);

   }

}
```

4 Save the file.

5 Create the file **decode.c**. Enter this code:

```
#include <stdio.h>

void decode_file(FILE *, FILE *);

int main()
{
   FILE *in_file, *out_file;
   char infile_name[81], outfile_name[81], input;

   printf("Enter the name of the file to decode: ");
   scanf("%s", infile_name);
   if ((in_file = fopen(infile_name, "r") ) == NULL)
   {
          printf("Error opening %s\n", infile_name);
          exit(0);
   }
   printf("Enter the output file name: ");
   scanf("%s", outfile_name);
   if ((out_file = fopen(outfile_name, "w") ) == NULL)
   {
          printf("Error opening %s\n", outfile_name);
          exit(0);
   }
   decode_file(in_file, out_file);
   printf("The file has been decoded.\n");
   fclose(in_file);
   fclose(out_file);
}
```

6 Save the file.

7 Create the file **decode_file.c**. Enter this code:

```
#include <stdio.h>

void decode_file(FILE *in_file, FILE *out_file)
{
   while (!feof(in_file))
   {
     char input;

     input = fgetc(in_file);
     input -= 10;
        fputc(input, out_file);
   }
}
```

8 Save the file. You are now ready to create the makefiles for both the encode and decode programs.

9 Enter the following code in the editor, and save it in the file **encode_make**:

```
encode: encode.o encode_file.o
     gcc encode.o encode_file.o -o encode

encode.o: encode.c
     gcc -c encode.c

encode_file.o: encode_file.c
     gcc -c encode_file.c
```

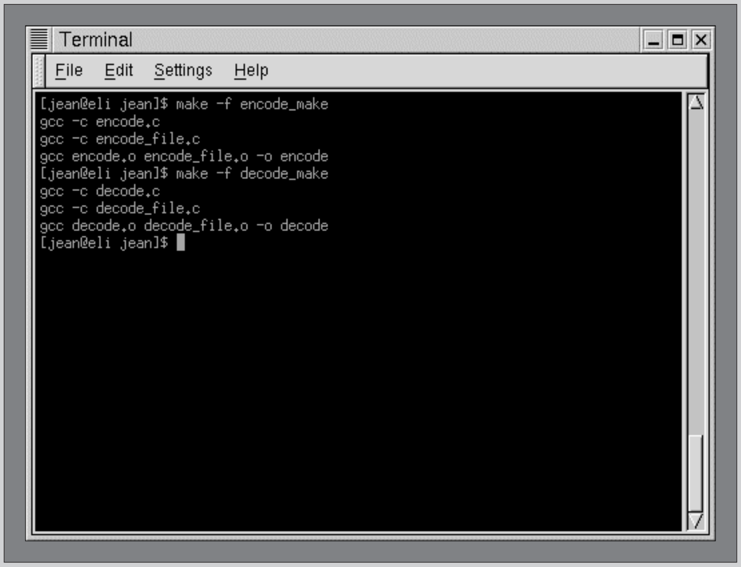
10 Create a file named **decode_make**, and enter this code:

```
decode: decode.o decode_file.o
     gcc decode.o decode_file.o -o decode

decode.o: decode.c
     gcc -c decode.c

decode_file.o: decode_file.c
     gcc -c decode_file.c
```

11 Save the file. You are ready to build the programs.

12 Type **make –f encode_make** and press **Enter**.

13 Type **make –f decode_make** and press **Enter**. Your screen should resemble Figure 10-14.

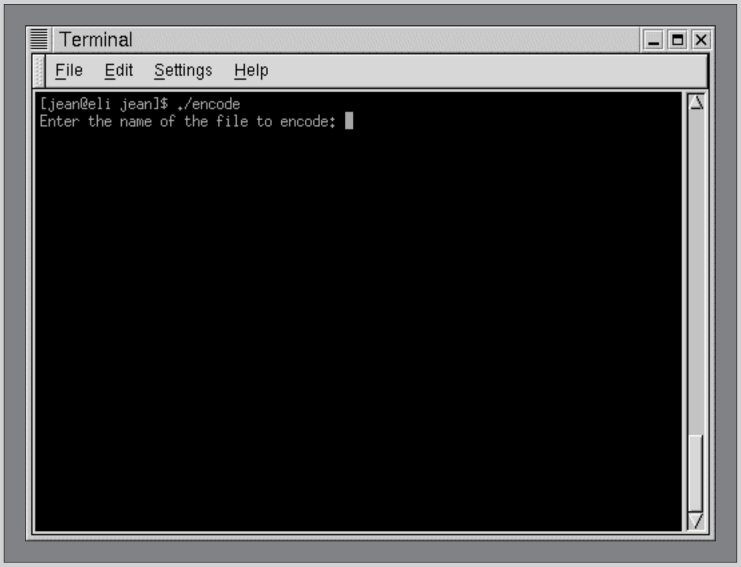


```
[jean@eli jean]$ make -f encode_make
gcc -c encode.c
gcc -c encode_file.c
gcc encode.o encode_file.o -o encode
[jean@eli jean]$ make -f decode_make
gcc -c decode.c
gcc -c decode_file.c
gcc decode.o decode_file.o -o decode
[jean@eli jean]$
```

Figure 10-14: Output of make commands

You will test the encode program by encrypting the testfile that you created in file I/O exercise. The file contains the string "This was created by a C program."

14 Type **./encode** and press **Enter**. Your screen appears similar to Figure 10-15.



```
[jean@eli jean]$ ./encode
Enter the name of the file to encode: 
```

Figure 10-15: Encode program

15 In response to the prompt, type **testfile** and press **Enter**.

16 The program now asks for the name of the output file.

17 Type **secret_file** and press **Enter**.

The contents of testfile have been encoded and stored in secret_file.

18 Use the **cat** command to look at the contents of secret_file. Your screen looks similar to Figure 10-16.

19 Run the decode program by typing **./decode** and pressing **Enter**.

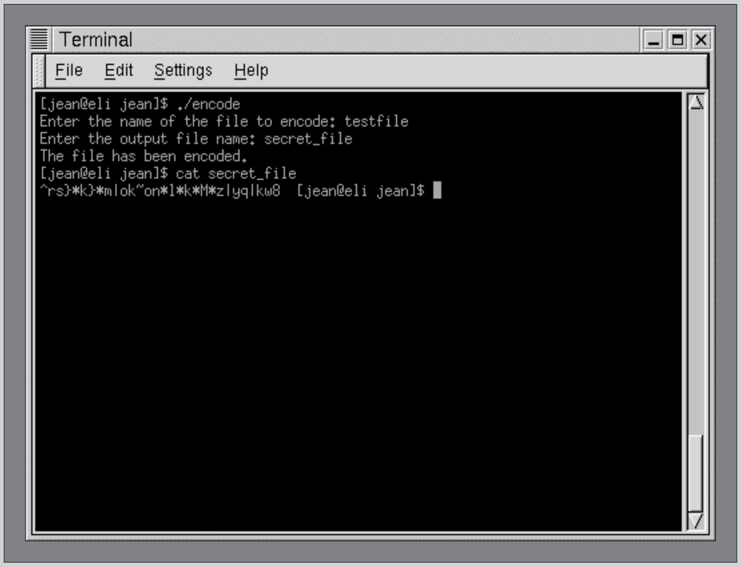
The program asks you to enter the name of the file to decode.

20 Type **secret_file** and press **Enter**.

Next the program asks you to enter the output file name.

21 Type **normal_file** and press **Enter**.

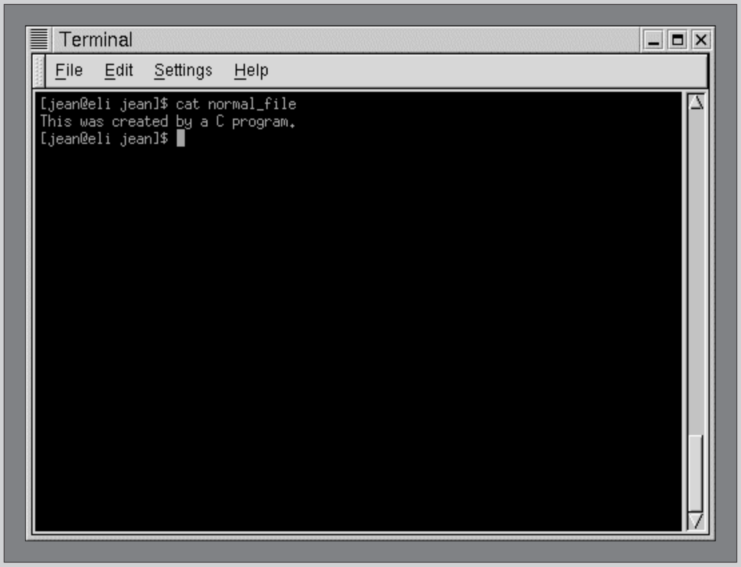
The contents of secret_file have been decoded and stored in normal_file.



```
[jean@eli jean]$ ./encode
Enter the name of the file to encode: testfile
Enter the output file name: secret_file
The file has been encoded.
[jean@eli jean]$ cat secret_file
^rs}*k}*mlok~on*l*k*M*zlyqlkw8  [jean@eli jean]$
```

Figure 10-16: Results of encode program

22 Use the **cat** command to look at the contents of normal_file. Your screen should look similar to Figure 10-17.



```
[jean@eli jean]$ cat normal_file
This was created by a C program.
[jean@eli jean]$
```

Figure 10-17: Contents of normal_file

You have now learned some fundamentals of programming in C, including working with files, using the make utility to maintain program source files, debugging your program, and using the encoding and decoding programs.

SUMMARY

- The C language concentrates on how best to create commands and expressions that can be elegantly formed from operators and operands.
- C programs often consist of separate source files called program modules that are compiled separately into object code and linked to the other object codes that make up the program.
- The C program structure begins with the execution of instructions located inside a main function that calls other functions that contain more instructions.
- The make utility is used to maintain the application's source files and the default make control file is called makefile.

REVIEW QUESTIONS

1. In general a make rule has a ________________.
  a. target, dependencies, and an action
  b. source, object, and executable
  c. header, a body, and a footer
  d. target and an action

2. The name of a C function can be recognized because it is followed by ________________.
  a. < >
  b. ( )
  c. ( );
  d. { }

3. In C, a block of instructions is enclosed inside ________________.
  a. < >
  b. ( )
  c. [ ]
  d. { }

4. The preprocessor reads the contents of a C source file, looking for statements that begin with ________________.
  a. $
  b. %
  c. #
  d. !

5. In an #include directive, the name of the standard library header files are enclosed with ________________.
   a. the less-than "<" and greater-than ">"
   b. double quotation marks " "
   c. left and right brackets [ ]
   d. left and right parentheses ( )
6. The unsigned and short modifiers may be applied to the ________________ data type(s).
   a. float and double
   b. int
   c. double
   d. int and char
7. In a C program, character constants are enclosed in ________________.
   a. ""
   b. #
   c. ()
   d. []
8. The ampersand (&) that precedes a variable name is called the ________________ operator.
   a. declarative-pointer
   b. address-of
   c. reference
   d. fixed-pointer
9. The C programming development tool used to facilitate compiling and maintaining the source code is called the ________________.
   a. precompiler
   b. preprocessor
   c. make utility
   d. memory manager
10. When you compile a C program, the compiler creates a binary file called the ________________ file.
    a. source
    b. executable
    c. link
    d. object

EXERCISES

1. Write a C program named myname.c that displays your name on the screen.
2. Write a C program named calc.c that allows you to enter seven numbers. The program should calculate and display the sum and average of the numbers.

3. Write a program named condays.c that asks you to enter a number of days. The program should convert the number of days to a number of weeks and a number of months. Display the values.
4. Write a program named num_table.c that displays a table of the numbers 1 through 20, with the squares and cubes of the numbers.

DISCOVERY EXERCISES

1. Rewrite your solution to Exercise 4 so a function named display_table() displays the table of numbers.
2. Rewrite your solution to Discovery Exercise 1 so the display_table() function accepts an argument. The argument is the starting value of the table. For example, if 5 is passed to the function as an argument, the function displays the values 5–25, along with their squares and cubes.
3. Rewrite your solution to Discovery Exercise 2 so the display_table() function is in a separate file from function main. Compile and link the files.
4. Create a makefile to compile and create a program for the calc.c program you created in Exercise 2.
5. Create a makefile to compile and execute your solution to Discovery Exercise 3.

LESSON B

objectives

In this lesson you will:

- Create a C++ program that performs screen output
- Create a C++ program to read a text file
- Create a C++ program with overload functions
- Create a C++ program that creates a new class object

C++ Programming in a UNIX environment

Introducing C++ Programming

C++ is a programming language developed by Bjarne Stroustrup at AT&T Bell Labs. It builds on the C language to add object-oriented capabilities. As a result, C++ is best learned after you have been programming in C for a while. With C++ you can do "more with less" after you learn its nuances. Functions, the building blocks of C programming, are incorporated in C++ with added dimensions such as **function overloading**, which makes the functions respond to more than one set of criteria and conditions.

C and C++ share many similarities. For example, programs in both languages start with the main() function and call other functions that include blocks of instructions enclosed within curly braces. Both languages also have similar source files. The C++ compiler readily accepts C language syntax and coding structures. For example, you can take the file encryption and decryption programs you created in Lesson A and fully compile them using the C++ compiler. Both languages fully support compiler directives such as #include and #define. The C++ compiler's name is **CC** for most UNIX versions and **g++** for Linux versions.

Note: One important distinction should be made about C++ programs. You can place your variable declarations anywhere inside the program, before or after the instructions. This is not true of C programs, in which program variables must precede all the instructions.

The major differences between the two languages become evident when you start using the C++ enhancements and class structures, which depart dramatically from standard C procedures. C follows procedural principles, whereas C++ primarily follows object-oriented programming principles while still allowing procedural programming methods. Procedural programming follows long-standing traditions that separate the data to be processed from the procedures that process. Procedural techniques require that the data fields be named and defined by data types (integers, characters, strings, floating decimals, and a variety of structures and arrays) before any processing begins. Object-oriented programming, on the other hand, allows the data to be described by name and type anywhere in the program. More significantly, C++ programs introduce objects as a new data class. **Objects** are a collection of data and a set of operations, called **methods**, that manipulate the data. Unlike standard C functions, C++ methods are part of the object they belong to, not the program.

Other more minor differences between C and C++ concern the name of the compiler (Linux calls the C++ compiler **g++**) and the suffix attached to a C++ source file, often **.C** or **.cpp**.

Creating a Simple C++ Program

To illustrate the similarity between C and C++, you will create a short program, simple.C, which displays a message on the screen exactly as the program simple.c does. The differences between the two languages start with the #include <iostream.h> instead of #include <stdio.h>. The only other difference is the use of the cout I/O stream object instead of printf.

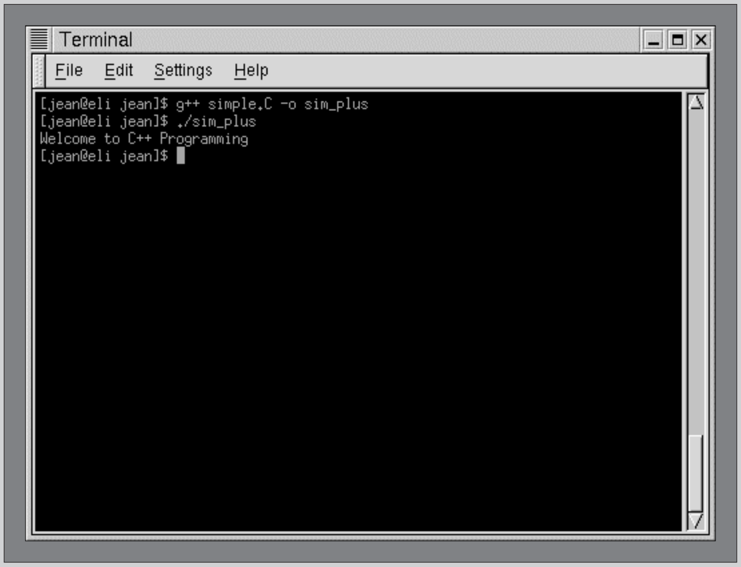
To write a C++ program:

1 Use the editor of your choice to create **simple.C**. Enter this code:

```
//=========================================================
// Program Name: simple.C
// By:           JQD
//Purpose:       First program in C++ showing how to
//               produce output.

//=========================================================

#include <iostream.h>

void main(void)
{
     cout << "Welcome to C++ Programming\n";
}
```

2 Use the C++ compiler to create a program called **sim_plus** by typing **g++ simple.C –o sim_plus** and pressing **Enter**.

3 Run sim_plus. Your screen looks similar to Figure 10-18.



```
[jean@eli jean]$ g++ simple.C -o sim_plus
[jean@eli jean]$ ./sim_plus
Welcome to C++ Programming
[jean@eli jean]$ 
```

Figure 10-18: Output of sim_plus program

Looking at the program, notice that C++ uses // to denote a comment line. (You can also use C's /* and */ to enclose comments in your C++ program.) Recall that comments help to identify and describe the program for all who need to review the program. Comments are ignored by the compiler and do not cause the computer to perform any action when the program runs.

Further, note that the standard library functions for I/O are found in the iostream.h instead of stdio.h as in the C program. The only other difference between the C and C++ programs is the use of cout in the C++ program.

To continue the comparison between C and C++, you'll next see how a C++ program reads and displays the information in a file.

Creating a C++ Program That Reads a Text File

You will learn further differences between C and C++ by entering the next C++ program, which reads a text file.

To create a C++ program that reads a text file:

1 Use the editor of your choice to create the file **fileread.C**. Enter this code:

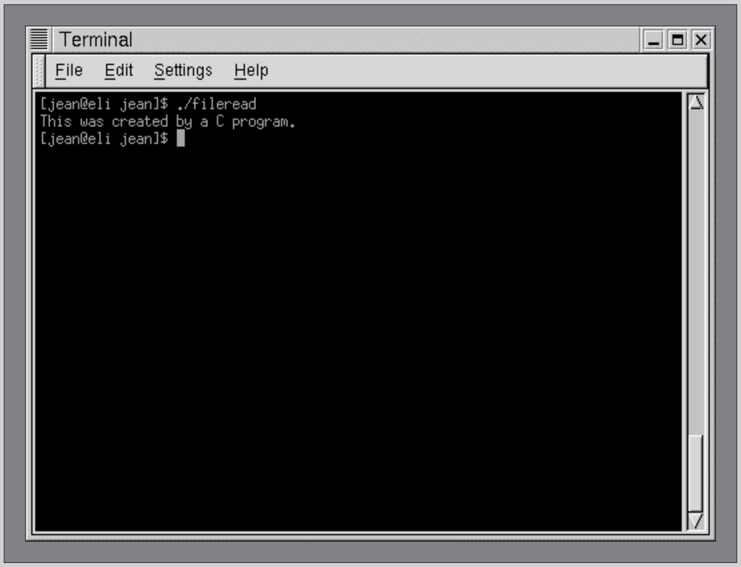
```
// A C++ file that reads the contents of a file.
#include <fstream.h>
```

```
void main(void)
{
     ifstream file("testfile");
     char record_in[256];

     if (file.fail())
          cout << "Error opening file.\n";
     else
     {
          while (!file.eof())
          {
               file.getline(record_in, sizeof(record_in));
               if (file.good())
                    cout << record_in << endl;
          }
     }
}
```

2 Save the file and exit the editor.

3 Compile the program and save the executable code in fileread.

4 Test the fileread program. Your screen should appear similar to Figure 10-19.



```
[jean@eli jean]$ ./fileread
This was created by a C program.
[jean@eli jean]$ 
```

Figure 10-19: Output of fileread program

There are several differences in the way C and C++ handle file operations. For example, the code:

```
ifstream file ("testfile");
```

tells the compiler to use the ifstream class to perform file input and output operations. The identifier *file* follows the class name. This statement is similar to the following C statement:

```
FILE *file;
```

Further, the file.fail() function is a part of the ifstream class and reports an invalid condition with the file access. The endl stream manipulator causes the screen output to skip a line.

The file.getline() function reads in a line from the file and stores it in the buffer record_in for subsequent processing. The file.good flag is a component of ifstream class and is used to determine if the record accessed contains data.

Now that you have an understanding of how C++ is similar to C, next you'll see how C++ provides additional enhancements.

How C++ Enhances C Functions

C++ created a way to define a function so that it can handle multiple sets of criteria, called **function overloading**. Whereas C functions are quite flexible, function overloading adds considerably to their overall use by expanding the function definition to accept varying kinds and numbers of parameters. During compilation, the C++ compiler determines which function to call based on the number and types of parameters the calling statement passes to the function. For example, in the next exercise you overload a function to access the system date in two different ways.

To use function overloading:

1 Use the editor of your choice to type the contents of datestuf.C:

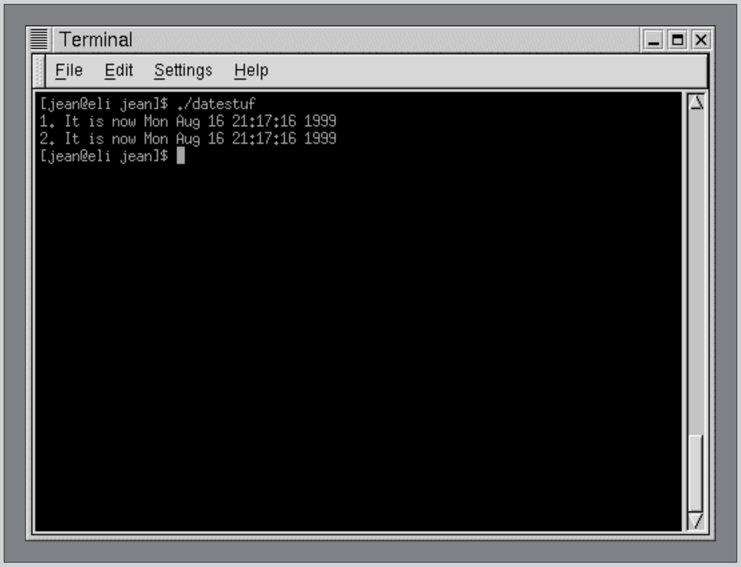
```
// Program name: datestuf.C
// Purpose: shows you two ways to access the system date.

#include <iostream.h>
#include <time.h>

void display_time(const struct tm *tim)
{
     cout << "1. It is now " << asctime(tim);
}

void display_time(const time_t *tim)
{
     cout << "2. It is now " << ctime(tim);
}
```

```
void main(void)
{
     time_t tim = time(NULL);
     struct tm *ltim = localtime(&tim);
     display_time(ltim);
     display_time(&tim);
}
```

2 Save the file and then exit editor.

3 Compile datestuff.C by typing **g++ datestuf.C -o datestuf** and pressing **Enter**.

4 Test the program. Your screen should be similar to Figure 10-20.



```
[jean@eli jean]$ ./datestuf
1. It is now Mon Aug 16 21:17:16 1999
2. It is now Mon Aug 16 21:17:16 1999
[jean@eli jean]$ 
```

Figure 10-20: Output of datestuf program

Notice how the same function name is used for the different calls to the two different date types that are contained in <time.h>. One is a structure; the other is not.

```
void display_time (const struct tm *tim)
void display_time (const time_t *tim)
```

The program is able to distinguish which function to use based on the type being passed to it.

`Display_time(ltim);`	Uses the structure type
`Display_time(&tim);`	Uses the time_t type

Now that you have learned the basic structure of a C++ program, you will learn to create object-oriented programs with the C++ class construct.

Setting Up a Class

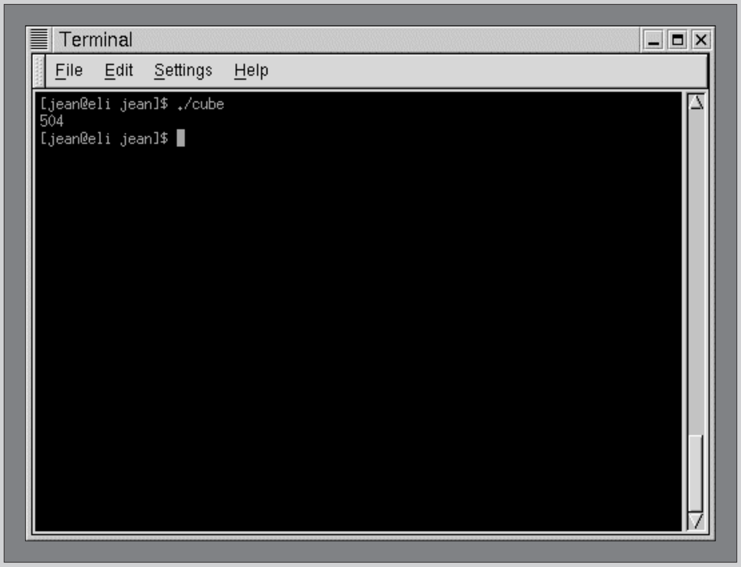
One of the more difficult concepts to grasp is the use of the C++ class data structure. A data structure lets you create abstract data types. An **abstract data type** is one defined by the programmer for a specific programming task.

You might begin by thinking of the class as made up of members that interrelate to make the class perform like an object rather than just a normal structure. Its methods (which are like C functions) are part of the class and considered behaviors of the class. The similarity between a class and a structure is that both store related data. You can use structures in C++ just as you do in C. However, in C++ you can and should use a class when your program performs specific operations on the data. For example, you can create a class for an object called cube when you want to compute the volume of any size cube:

```
//============================================================

// Program Name: cube.C
// Purpose: Show how to set up a class. The class is called
//          cube, and computes the volume of a cube.

//============================================================

#include <iostream.h>

//---- cube class
class Cube
{
     int height, width, depth; // private data members
public:
     // ----- constructor
     Cube(int ht, int wd, int dp)
          { height = ht; width = wd; depth = dp; }
     // ----- member function
     int volume()
          { return height * width * depth; }
};

void main(void)
{
     Cube thiscube(7, 8, 9);  // declare a Cube
     cout << thiscube.volume() << "\n";  // Compute &
display volume
}
```

To compute the volume of any size cube:

1. Use an editor to type in cube.C shown above, and then save the file.
2. Compile cube.C and store the executable code in the file cube.
3. Test cube. Your screen should look similar to Figure 10-21.



```
[jean@eli jean]$ ./cube
504
[jean@eli jean]$ 
```

Figure 10-21: Output of cube program

In looking at the program, the line class cube tells the compiler that you are declaring a new class called cube. The variables within the class are called private data members and can only be accessed by members of this class. If you create objects in your program of type cube, then they can access the cube's private data members: height, width, and depth variables.

Constructors and other member functions can be defined outside, as well as inside, of a class definition. Unlike a structure, whose members are all accessible to a program, a class can have members that the program can directly access using the dot (.) operator (public members) and other members that the program cannot access directly (private members). To access the private data and methods, the program must call the public methods.

In this lesson you have learned the basic differences between C and C++ programs. You have written C++ programs that perform screen and keyboard I/O, as well as file operations. In addition, you have created simple programs with class objects.

SUMMARY

- The major difference between C and C++ is that C follows procedural principles and C++ primarily follows object-oriented programming.
- The standard stream library used by C++ is iostream.h.
- C++ provides two statements for standard input and standard output: cin and cout respectively. These are defined in the class libraries contained in <iostream.h>.
- C++ offers a way to define a function so that it can handle multiple sets of criteria. This function is called overloading.
- endl skips a line like "\n" does in the C language.
- A C++ class is made up of members that interrelate to make it perform like an object rather than just a normal structure.
- You should use a class in C++ when your program performs specific operations on the data.

REVIEW QUESTIONS

1. C++ introduces a stream object for displaying output, which is called ________________.
   a. cout
   b. cin
   c. cerr
   d. printf

2. C++'s stream object that interacts between the user and computer to handle keyboard input is ________________.
   a. cin
   b. cout
   c. cerr
   d. getchar

3. In C++, what do you call the structure that consists of data members and methods?
   a. stream
   b. class
   c. object
   d. array

4. A class can have members that the program can directly access using the dot (.) operator. These accessible members are identified with the ________________ access specifier.
   a. private
   b. public
   c. inline
   d. offline
5. You can enter comments in the C++ source code by using ________________.
   a. two slashes "//" preceding the comment
   b. the C /* and */ enclosures
   c. preceding the comment with ##
   d. both a and b
6. Which of these contains incorrect syntax?
   a. cout << "My Name is: "; // prompt for name
   b. cout >> "My Name is: "; // prompt for name
   c. cout << "My Name is: " << endl; // prompt for name
   d. cout << "My Name is:\n"; // prompt for name
7. Which of these is using incorrect syntax?
   a. cin >> integer1; // read an integer
   b. cin >> name; // read a string field
   c. cin << integer1; // read an integer
   d. cin >> letter; // read a char
8. The ________________ initializes object members automatically every time the program creates a class instance.
   a. static function
   b. constructor function
   c. private variables
   d. object assignment

E X E R C I S E S

1. Write a small C++ program called nameaddr.C to display your name and address.
2. Refer to the C program, keyboard.c, presented in Lesson A, and write the equivalent program, called keyboard.C, in C++. Compile and test it.
3. Write a C++ program that does the following:

   Asks the user for values to be stored in the variables E and R.
   Multiplies E times R and stores the result in the variable I.
   If I is greater than 10, prints the message, "Value exceeds upper limit." If I is less than 1, prints the message, "Value does not meet the lower limit."

DISCOVERY EXERCISES

1. Create a C++ version of the num_table.c program you created in Exercise 4 of Lesson A.
2. Modify the program you wrote for Discovery Exercise 1 so it uses a different type of loop. For example, if the program now uses a for loop, rewrite it so it uses a while loop.
3. Create a C++ program with a class named circle. The class should have the following member variables: radius, diameter, and area.

   The constructor should accept one argument: the circle's radius. The constructor should calculate the diameter of the circle as radius * 2, and the area as 3.1416 * radius * radius. These values should be stored in the class's member variables. In addition, the class should have a member function that returns the circle's area, a member function that returns the circle's diameter, and a member function that returns the circle's area. Demonstrate the class in a simple program.